KIP-516: Topic Identifiers

Status

Current state: *Accepted*

Discussion thread: https://lists.apache.org/thread.html/7efa8cd169cadc7dc9cf86a7c0dbbab1836ddb5024d310fcebacf80c@%3Cdev.kafka.apache.org%3E

JIRA: ⚠ Unable to render Jira issues macro, execution error.

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Today, Kafka uniquely identifies a topic by its name. This is generally sufficient however there are flaws with this scheme when a topic is deleted and recreated with the same name. Kafka currently attempts to prevent issues resulting from stale topics by ensuring a topic is fully deleted from all replicas before completing a deletion. This solution is imperfect, as it is possible for partitions to be reassigned away from brokers while they are down and there are no guarantees that this state will ever be cleaned up.

When a topic deletion is performed the controller must wait for all brokers to delete their local replicas. This blocks creation of a topic with the same name as a deleted topic until all replicas have successfully deleted the topic's data. This can mean that downtime for a single broker can effectively cause a complete outage for everyone producing/consuming to that topic name if a topic cannot be recreated without manual intervention.

Topic IDs aim to address this issue by associating a truly unique ID with each topic, ensuring a newly created topic with a previously used name cannot be confused with a previous topic with that name.

Topic IDs solve several additional problems:

1. Renaming topics becomes feasible (although there may still be some complexity with the need to support the old name for a while as part of migration, etc.) Renaming topics may seem minor, but it will be difficult to have hierarchical topics without having some kind of renaming support.
2. We can eventually get rid of the "deleting" state for topics. If a broker is down but there is some topic data there that is no longer relevant, it won't cause problems later on. It can be deleted when the broker rejoins the cluster and realizes that the relevant topic ID is not present any more. We gain some additional safety where stale/deleted replicas may currently interact with live ones.
3. Sending 16 byte UUIDs instead of Strings over Kafka RPCs can be smaller. A string is 2 bytes plus the data, whereas the UUID is fixed 16 bytes. For any topic name with more than than 14 single byte characters (16 bytes serialized), UUIDs will be smaller. They will also be faster to compare and more friendly to the garbage collector.
4. They will provide a true measure of topic uniqueness across clusters. This may be important in multi-cluster Kafka deployments where additional safety and debuggability is desired.

Overall, topic IDs provide a safer way for brokers to replicate topics without any chance of incorrectly interacting with stale topics with the same name. By preventing such scenarios, we can simplify a number of other interactions such as topic deletes which are currently more complicated and problematic than necessary.

Public Interfaces

Uuid

A new Uuid class will be exposed under /org/apache/kafka/common

```
/*
 * This class defines an immutable universally unique identifier (Uuid). It represents a 128-bit value.
 * More specifically, the random Uuids in this class are variant 2 (Leach-Salz) version 4 Uuids.
 * This definition is very similar to java.util.UUID. The toString() method prints
 * using the base64 string encoding. Likewise, the fromString method expects a base64 string encoding.
 */
public class Uuid {

  /**
   * A Uuid where all bits are zero. It represents a null or empty Uuid.
   */
  public static final Uuid ZERO_UUID

  /**
   * Constructs a 128-bit type 4 Uuid where the first long represents the the most significant 64 bits
   * and the second long represents the least significant 64 bits.
   */
  public Uuid(long mostSigBits, long leastSigBits)

  /**
   * Static factory to retrieve a type 4 (pseudo randomly generated) Uuid.
   */
  public static Uuid randomUuid()

  /**
   * Returns the most significant bits of the Uuid's 128 bit value.
   */
  public long getMostSignificantBits()

  /**
   * Returns the least significant bits of the Uuid's 128 bit value.
   */
  public long getLeastSignificantBits()

  /**
   * Returns true iff the obj is another Uuid with the same value.
   */
  public boolean equals(Object obj)

  /**
   * Returns a hash code for this Uuid
   */
  public int hashCode()

  /**
   * Returns a base64 string encoding of the Uuid.
   */
  public String toString()

  /**
   * Creates a Uuid based on a base64 string encoding used in the toString() method.
   */
  public static Uuid fromString(String str)

}


Some public api will be added to org.apache.kafka.common.Cluster

/**
* An immutable representation of a subset of the nodes, topics, and partitions in the Kafka cluster.
*/
public class Cluster {

  /**
   * Get All topicIds in the cluster, similar to topics()
   */
  public Collection<Uuid> topicIds()

  /**
   * Get the topicId of a topic, Uuid.ZERO_UUID is returned topicId doesn't exists.
   */
 public Uuid topicId(String topic)



}
```

Additionally, it may be dangerous to use older versions of Kafka tools with new broker versions when using their `--zookeeper` flags. Use of older tools in this way is not supported today.

Proposed Changes

Topic IDs will be represented with 128 bit v4 UUIDs. A UUID with all bits as 0 will be reserved as a null UUID as the Kafka RPC protocol does not allow for nullable fields. When printed or stored as a string, topic IDs will be converted to base64 string representation.

On handling a CreateTopicRequest brokers will create the topic znode under */brokers/topics/[topic],* as usual.

The znode value will now contain an additional topic ID field, represented as a base64 string in the "*id"* field, and the schema version will be bumped to version 3.

```
Schema:
{ "fields" :
    [ {"name": "version", "type": "int", "doc": "version id},
      {"name": "id", "type": "string", "doc": option[Uuid]},
      {"name": "partitions",
       "type": {"type": "map",
                "values": {"type": "array", "items": "int", "doc": "a list of replica ids"},
                "doc": "a map from partition id to replica list"},

      {"name": "adding_replicas",
       "type": {"type": "map",
                "values": {"type": "array", "items": "int", "doc": "a list of replica ids"},
                "doc": "a map from partition id to a list of replicas to add in a pending reassignment"},

      {"name": "removing_replicas",
       "type": {"type": "map",
                "values": {"type": "array", "items": "int", "doc": "a list of replica ids"},
                "doc": "a map from partition id to a list of replicas to remove in a pending reassignment"},
      }
    ]
}

Example:
{
  "version": 3,
  "id": "b8tRS7h4TJ2Vt43Dp85v2A",
  "partitions": {"0": [0, 1, 3] },
  "adding_replicas": {},
  "removing_replicas": {}
}
```

The controller will maintain local in-memory state containing a mapping from topic name to topic ID. On controller startup, the topic ID will automatically be loaded into memory along with the topics and partitions. A random UUID will be generated on topic creation or on migration of an existing topic without topic IDs.

The controller will supply topic IDs for all topic partitions to brokers by sending LeaderAndIsrRequest(s) that contain the topic IDs for all partitions contained in the request.

Requests to describe topics will return a result containing TopicDescriptions with topic IDs for each topic

Protocol Changes

Removal of Topic Names from Request and Responses

It is unnecessary to include the name of the topic in the following Request/Response calls:
LeaderAndIsr (Response only)
StopReplica
Fetch
List Offsets
OffsetForLeader
Produce
Vote
BeginQuorumEpoch
EndQuorumEpoch

Including the topic name in the request may make it easier to debug when issues arise, as it will provide more information than the topic ID alone. However, it will also bloat the protocol (especially relevant for FetchRequest), and if they are incorrectly used it may prevent topic renames from being easily implemented in the future. For these reasons, the topic name field has been removed.

LeaderAndIsr

LeaderAndIsrRequest v5

```
LeaderAndIsr Request (Version: 5) => controller_id controller_epoch broker_epoch type* [topic_states]
[live_leaders]
  controller_id => INT32
  controller_epoch => INT32
  broker_epoch => INT64
  type* => INT8
  topic_states => topic topic_id* [partition_states]
    topic => STRING
    topic_id* => UUID
    partition_states => partition controller_epoch leader leader_epoch [isr] zk_version [replicas] is_new
      partition => INT32
      controller_epoch => INT32
      leader => INT32
      leader_epoch => INT32
      isr => INT32
      zk_version => INT32
      replicas => INT32
      is_new => BOOLEAN
  live_leaders => id host port
    id => INT32
    host => STRING
    port => INT32
```

LeaderAndIsrRequest v5 adds the topic ID to the topic_states field, and an enum *type* to denote the type of LeaderAndIsrRequest. Currently, the first LeaderAndIsrRequest sent to a broker by a controller contains all TopicPartitions that a broker is a replica for. We will formalize this behavior by also including a *type* enum to denote the type of LeaderAndIsrRequest. IBP will be used to determine whether this new form of the request will be used. For older requests, we will ignore this field and default to previous behavior.

value	enum	description
1	INCREMENTAL	A LeaderAndIsrRequest that is not guaranteed to contain all topic partitions assigned to a broker.
2	FULL	A full LeaderAndIsrRequest containing all partitions the broker is a replica for.

When *type* = *FULL*, the broker is able to reconcile its local state on disk with the request. Any partition not contained in this request and present on local disk can be staged for deletion. There are two such types of stale request. In both cases the broker's topic will be staged for deletion.

1. The TopicPartition is not present in the LeaderAndIsrRequest.

2. The TopicPartition is contained in the request, but the topic ID that does not match the local topic partition stored on the broker.

Reconciliation may also be necessary if *type = INCREMENTAL* and the topic ID set on a local partition does not match the topic ID contained in the request. A TopicPartition with the same name and a different topic ID by implies that the local topic partition is stale, as the topic must have been deleted to create a new topic with a different topic ID. This is similar to the type 2 stale request above, and the topic will be staged for deletion.

In the case where the topic ID in the request does not match the topic ID in the log (in either FULL or INCREMENTAL requests), we will also return a new exception `INCONSISTENT_TOPIC_ID`.
This exception will be used for when the topic ID in memory does not match the topic ID in the request.

Deletion

Deletion of stale partitions triggered by LeaderAndIsrRequest(s) will take place by:

1. Logging at WARN level all partitions that will be deleted and the time that they will be be deleted at.
2. Move the partition's directory to *log.dir/deleting/{topic_id}_{partition}*
3. Schedule deletion from disk with a delay of **delete.topic.delay.ms** ms. This will clear the *deleting* directory of the partition's contents.

LeaderAndIsrResponse v5

```
LeaderAndIsr Response (Version: 5) => error_code [topics]
  error_code => INT16
  topics => topic_id* [partitions]
   topic_id* => UUID
    partitions => partition error_code
      partition => INT32
      error_code => INT16
```

The topic name field has been removed.

StopReplica

StopReplicaRequest v4

```
StopReplica Request (Version: 4) => controller_id controller_epoch broker_epoch delete_partitions [topic_states]
  controller_id => INT32
  controller_epoch => INT32
  broker_epoch => INT64
  delete_partitions => BOOLEAN
  topic_states => topic_id* [partitions_states]
    topic_id* => UUID
    partition_states => partition_id leader_epoch delete_partition
      partition_id => INT32
      leader_epoch => INT32
      delete_parition => BOOLEAN
```

StopReplicaResponse v4

```
StopReplica Response (Version: 4) => error_code [topics]
  error_code => INT16
  topics => topic_id* [partitions]
    topic_id* => UUID
    partitions => partition error_code
      partition => INT32
      error_code => INT16
```

Fetch

To avoid issues where requests are made to stale partitions, a topic_id field will be added to fence reads from deleted topics. Note that the leader epoch is not sufficient for preventing these issues, as the partition leader epoch is reset when a topic is deleted and recreated. To reduce the size of the request and response, the topic name field has been removed.

FetchRequest v13

```
Fetch Request (Version: 13) => replica_id max_wait_time min_bytes max_bytes isolation_level session_id
session_epoch [topics] [forgotten_topics_data] rack_id
  replica_id => INT32
  max_wait_time => INT32
  min_bytes => INT32
  max_bytes => INT32
  isolation_level => INT8
  session_id => INT32
  session_epoch => INT32
  topics => topic_id* [partitions]
    topic_id* => UUID
    partitions => partition current_leader_epoch fetch_offset last_fetched_epoch log_start_offset
partition_max_bytes
      partition => INT32
      current_leader_epoch => INT32
      fetch_offset => INT64
      last_fetched_epoch => INT32
      log_start_offset => INT64
      partition_max_bytes => INT32
  forgotten_topics_data => topic_id* [partitions]
    topic_id* => UUID
    partitions => INT32
  rack_id => STRING
```

FetchResponse v13

```
Fetch Response (Version: 13) => throttle_time_ms error_code session_id [responses]
  throttle_time_ms => INT32
  error_code => INT16
  session_id => INT32
  responses => topic_id* [partition_responses]
    topic_id* => UUID
    partition_responses => partition error_code high_watermark last_stable_offset log_start_offset
diverging_epoch current_leader [aborted_transactions] preferred_read_replica record_set
        partition => INT32
        error_code => INT16
        high_watermark => INT64
        last_stable_offset => INT64
        log_start_offset => INT64
        diverging_epoch => epoch end_offset
          epoch => INT32
          end_offset => INT64
        current_leader => leader_id leader_epoch
          leader_id => INT32
          leader_epoch => INT32
        aborted_transactions => producer_id first_offset
          producer_id => INT64
          first_offset => INT64
        preferred_read_replica => INT32
        record_set => RECORDS
```

ListOffsets

To avoid issues where requests are made to stale partitions, a topic_id field will be added to fence reads from deleted topics.

ListOffsetRequest v6

```
ListOffset Request (Version: 6) => replica_id isolation_level [topics]
  replica_id => INT32
  isolation_level => INT8
  topics => topic_id* [partitions]
    topic_id* => UUID
    partitions => partition current_leader_epoch timestamp
      partition => INT32
      current_leader_epoch => INT32
      timestamp => INT64
```

ListOffsetResponse v6

```
ListOffset Response (Version: 6) => throttle_time_ms [responses]
  throttle_time_ms => INT32
  responses => topic_id* [partition_responses]
    topic_id* => UUID
    partition_responses => partition error_code timestamp offset leader_epoch
      partition => INT32
      error_code => INT16
      timestamp => INT64
      offset => INT64
      leader_epoch => INT32
```

OffsetForLeader

To avoid issues where requests are made to stale partitions, a topic_id field will be added to fence reads from deleted topics.

OffsetForLeaderRequest v4

```
OffsetForLeaderEpoch Request (Version: 4) => replica_id [topics]
  replica_id => INT32
  topics => topic_id* [partitions]
    topic_id* => UUID
    partitions => partition current_leader_epoch leader_epoch
      partition => INT32
      current_leader_epoch => INT32
      leader_epoch => INT32
```

OffsetForLeaderResponse v4

```
OffsetForLeaderEpoch Response (Version: 4) => throttle_time_ms [topics]
  throttle_time_ms => INT32
  topics => topic_id* [partitions]
    topic_id* => UUID
    partitions => error_code partition leader_epoch end_offset
      error_code => INT16
      partition => INT32
      leader_epoch => INT32
      end_offset => INT64
```

UpdateMetadata

UpdateMetadata should also include the topic ID.

UpdateMetadataRequest v7

```
UpdateMetadata Request (Version: 7) => controller_id controller_epoch broker_epoch [ungrouped_partition_states]
[topic_states] [live_brokers]
    controller_id => INT32
    controller_epoch => INT32
    broker_epoch => INT64
    ungrouped_partition_states => UpdateMetadataPartitionState
    topic_states => topic_name topic_id* [partition_states]
        topic_name => STRING
        topic_id* => UUID
        partition_states => UpdateMetadataPartitionState
    live_brokers => id v0_host v0_port [endpoints] rack
        id => INT32
        v0_host => STRING
        v0_port => INT32
        endpoints => port host listener security_protocol
            port => INT32
            host => STRING
            listener => STRING
            security_protocol => INT16
        rack => STRING
```

Produce

Swapping a the topic name for the topic ID will cut down on the size of the request.

ProduceRequest v9

```
Produce Request (Version 9) => transactional_id acks timeout_ms [topics]
    transactional_id => STRING
    acks => INT16
    timeout_ms => INT32
    topics => topic_id* [partitions]
        topic_id* => UUID
        partitions => partition_index records
            partition_index => INT32
            records => BYTES
```

ProduceResponse v9

```
Produce Response (Version 9) => [responses] throttle_time_ms
    responses => topic_id* [partitions]
        topic_id* => UUID
        partitions => partition_index error_code base_offset log_append_time_ms log_start_offset
[record_errors] error_message
            partition_index => INT32
            error_code => INT16
            base_offset => INT64
            log_append_time_ms => INT64
            log_start_offset => INT64
            record_errors => batch_index batch_index_error_message
                batch_index => INT32
                batch_index_error_message => STRING
            error_message => STRING
    throttle_time_ms =>  INT32
```

DeleteTopics

With the addition of topic IDs and the changes to LeaderAndIsrRequest described above, we can now make changes to topic deletion logic that will allow topics to be immediately considered deleted, regardless of whether all replicas have responded to a DeleteTopicsRequest.

When the controller receives a DeleteTopicsRequest, if the IBP is >= MIN_TOPIC_ID_VERSION it will delete the */brokers/topics/[topic]* znode payload and immediately reply to the DeleteTopicsRequest with a successful response. At this point, the topic is considered deleted, and a topic with the same name can be created.

Although the topic is safely deleted at this point, it must still be garbage collected. To garbage collect, the controller will then send StopReplicaRequest(s) to all brokers assigned as replicas for the deleted topic. For the most part, deletion logic can be maintained between IBP versions, with some differences in responses and cleanup in ZooKeeper. Both formats must still be supported, as the IBP may not be bumped right away and deletes may have already been staged before the IBP bump occurs.

The updated controller's delete logic will:

1. Collect deleted topics:
    a. Old format: */admin/delete_topics* pulling the topic state from */brokers/topics/[topic]*.
    b. New in-memory topic deletion states from received DeleteTopicsRequest(s)
2. Remove deleted topics from replicas by sending StopReplicaRequest V3 before the IBP bump using the old logic, and using V4 and the new logic with topic IDs after the IBP bump.
3. Finalize successful deletes:
    a. For */admin/delete_topics* deletes, we may need to respond to the TopicDeleteRequest. We can also delete the topic znode at */admin/delete_topics/[topic]* and */brokers/topics/[topic]*.
    b. For deletes for topics with topic IDs, remove the topic from the in memory topic deletion state on the controller.
4. Any unsuccessful StopReplicaRequest(s) will be retried after retryMs, starting from 1) and will be maintained in memory.

This leads to the question of what should be done if the controller never receives a successful response from a replica for a StopReplicaRequest. Under such a scenario it is still safe to stop retrying after a reasonable number of retries and time. Given that LeaderAndIsrRequest v5 includes a *type* flag, allowing for *FULL* requests to be identified, any stale partitions will be reconciled and deleted by a broker on startup upon receiving the initial LeaderAndIsrRequest from the a controller. This condition is also safe if the controller changes before the StopReplicaRequest(s) succeed, as the new controller will send a *FULL* LeaderAndIsrRequest on becoming the leader, ensuring that any stale partitions are cleaned up.

Immediate delete scenarios

Stale reads

1. Broker B1 is a leader for topic partition A_p0_id0
2. Topic A id0 is deleted.
3. Topic A id1 is created.
4. Broker B1 has not yet received a new LeaderAndIsrRequest, nor a StopReplicaRequest for topic partition A_p0_id0
5. Broker B2 has received a LeaderAndIsrRequest for topic partition A_p0 _id0, and starts fetching from B1.

Inclusion of topic IDs in FetchRequest/ListOffsetRequest/OffsetsForLeaderEpochRequest(s) ensure that this scenario is safe. By adding the topic ID to these request types, any request to stale partitions will not be successful.

Stale state

1. Broker B1 is a replica for A_p0_id0.
2. Topic A id0 is deleted.
3. B1 and has not does not receive a StopReplicaRequest for A_p0_id0.
4. Topic A id1 is created.
5. Broker B1 receives a LeaderAndIsrRequest containing partition A_p0_id1.

When this occurs, we will close the Log for A_p0_id0, and move A_p0_id0 to the *deleting* directory as described in the LeaderAndIsrRequest description above.

Storage

Partition Metadata file

To allow brokers to resolve the topic name under this structure, a metadata file will be created at *logdir/partitiondir/partition.metadata*.

This metadata file will be human readable, and will include:

- Metadata schema version (schema_version: int32)
- Topic ID (id: Uuid)

This file will be plain text (key/value pairs).

```
version: 0

topic_id: 46bdb63f9e8d4a38bf7bee4eb2a794e4
```

One important use for this file is the current directory structure does not allow us to reload the broker's view of topic ID on startup (perhaps after a failure). It is necessary to persist this file to disk so this information can be reloaded.

It will be easy to update the file to include more fields in the future. This may assist with tooling purposes like mapping topic IDs to topic names.

In the JBOD mode, a partition's data can be moved from one disk to another. The partition metadata file would be copied during this process.

Tooling

kafka-topics.sh --describe will be updated to include the topic ID in the output. A user can specify a topic name to describe with the --topic parameter, or alternatively the user can supply a topic ID with the --topic_id parameter

Configuration

The following configuration options will be added:

Option	Unit	Default	Description
delete.topic.delay.ms	ms	14400 (4 hours)	The minimum amount of time to wait before removing a deleted topic's data on every broker

AdminClient Support

Access to topic IDs from the AdminClient will make it easier for users to obtain topics' topic IDs. It can also ensure correctness when deleting topics. This will require some changes to public APIs and protocols

TopicCollection

One change to help with the transition from defining topics by names to defining them by IDs is a new class that can represent a collection of topics by name or ID. This class can be passed in to methods that support identifying topics by either identifier–like describe and delete below. This will be found in the common package.

```
/**
 * A class used to represent a collection of topics. This collection may define topics by topic name
 * or topic ID. Subclassing this class beyond the classes provided here is not supported.
 */
public abstract class TopicCollection {

    private TopicCollection() {}

    /**
     * @return a collection of topics defined by topic ID
     */
    public static TopicIdCollection ofTopicIds(Collection<Uuid> topics);

    /**
     * @return a collection of topics defined by topic name
     */
    public static TopicNameCollection ofTopicNames(Collection<String> topics);

    /**
     * A class used to represent a collection of topics defined by their topic ID.
     * Subclassing this class beyond the classes provided here is not supported.
     */
    public static class TopicIdCollection extends TopicCollection {

        /**
         * @return A collection of topic IDs
         */
        public Collection<Uuid> topicIds();
    }

    /**
     * A class used to represent a collection of topics defined by their topic name.
     * Subclassing this class beyond the classes provided here is not supported.
     */
    public static class TopicNameCollection extends TopicCollection {

        /**
         * @return A collection of topic names
         */
        public Collection<String> topicNames();
    }
}
```

CreateTopics

Upon creation of a topic, the topic ID will be included in the TopicMetadataAndConfig which is included in CreateTopicsResult. It can be accessed through a method in CreateTopicsResult or the TopicMetadataAndConfig object.

CreateTopicsResult

```
public class CreateTopicsResult {

  public KafkaFuture<Uuid> topicId(String topic)

...

  public static class TopicMetadataAndConfig {

    TopicMetadataAndConfig(Uuid topicId, int numPartitions, int replicationFactor, Config config)

    public Uuid topicId()

}
```

The protocol for CreateTopicsResponse will also need a slight modification.

CreateTopicsResponse v7

```
CreateTopics Response (Version: 7) => throttle_time_ms [topics]
  throttle_time_ms => INT32
  topics => name topic_id* error_code error_message topic_config_error_code num_partitions replication_factor
[configs]
    name => STRING
    topic_id* => UUID
    error_code => INT16
    error_message => STRING
    topic_config_error_code => INT16
    num_partitions => INT32
    replication_factor => INT16
    configs => name value read_only config_source is_sensitive
      name => STRING
      value => STRING
      read_only => BOOL
      config_source => INT8
      is_sensitive => BOOL
```

Describe Topics

There are two use cases we want to support. 1) Obtaining topic IDs when asking to describe topics and 2) supplying topic IDs to get a description of the topics

For use case (1), we need to modify TopicDescription and MetadataResponse

TopicDescription

```
/**
* Create an instance with the specified parameters.
*
* @param name The topic name
* @param internal Whether the topic is internal to Kafka
* @param partitions A list of partitions where the index represents the partition id and the element contains
*                   leadership and replica information for that partition.
* @param authorizedOperations authorized operations for this topic, or null if this is not known.
* @param topicId Unique value that identifies the topic
*
*/
public TopicDescription(String name, boolean internal, List<TopicPartitionInfo> partitions,
    Set<AclOperation> authorizedOperations, Uuid topicId)



/**
* A unique identifier for the topic.
*/
public Uuid topicId()
```

MetadataResponse v10

```
Metadata Response (Version: 10) => throttle_time_ms [brokers] cluster_id controller_id [topics]
cluster_authorized_operations
    throttle_time_ms => INT32
    brokers => node_id host port rack
        node_id => INT32
        host => STRING
        port => INT32
        rack => STRING
    cluster_id => STRING
    controller_id => INT32
    topics => error_code name topic_id* is_internal [partitions] topic_authorized_operations
        error_code => INT16
        name => STRING
        topic_id* => UUID
        is_internal => BOOL
        partitions => error_code partition_index leader_id leader_epoch [replica_nodes] [isr_nodes]
[offline_replicas]
            error_code => INT16
            partition_index => INT32
            leader_id => INT32
            leader_epoch => INT32
            replica_nodes => INT32
            isr_nodes => INT32
            offline_replicas => INT32
        topic_authorized_operations => INT32
    cluster_authorized_operations => INT32
```

When topic IDs are supported, the response will contain both the topic name and the topic ID.

For use case (2), new methods will need to be added to the Admin interface and KafkaAdminClient

Admin and KafkaAdminClient

```
default DescribeTopicsResult describeTopics(TopicCollection topics);
```

```
DescribeTopicsResult describeTopics(TopicCollection topics, DescribeTopicsOptions options);
```

We also plan to deprecate the old methods in a future release. There are changes to DescribeTopicsResult and deprecation of some of its methods

```
public class DescribeTopicsResult {

    protected DescribeTopicsResult(Map<Uuid, KafkaFuture<Void>> topicIdFutures, Map<String, KafkaFuture<Void>>
nameFutures);

    protected static DescribeTopicsResult ofTopicIds(Map<Uuid, KafkaFuture<Void>> topicIdFutures);

    protected static DescribeTopicsResult ofTopicNames(Map<String, KafkaFuture<Void>> nameFutures);

   /**
    * @return a map from topic IDs to futures which can be used to check the status of
    * individual topics if the describeTopics request used topic IDs. Otherwise return null.
    */
    public Map<Uuid, KafkaFuture<Void>> topicIdValues()

   /**
    * @return a map from topic names to futures which can be used to check the status of
    * individual topics if the describeTopics request used topic names. Otherwise return null.
    */
    public Map<String, KafkaFuture<Void>> topicNameValues()

   @Deprecated
   /**
    * @return a map from topic names to futures which can be used to check the status of
    * individual topics if the describeTopics request used topic names. Otherwise return null.
    */
    public Map<String, KafkaFuture<Void>> values()

   /**
    * @return a future which succeeds only if all the topic descriptions succeed and the describeTopics
    * request used topic IDs.
    */
    public KafkaFuture<Map<Uuid, TopicDescription>> allTopicIds()

   /**
    * @return a future which succeeds only if all the topic descriptions succeed and the describeTopics
    * request used topic names.
    */
    public KafkaFuture<Map<String, TopicDescription>> allTopicNames()

   @Deprecated
  /**
   * Return a future which succeeds only if all the topic descriptions succeed and the describeTopics
   * request used topic names.
   */
   public KafkaFuture<Void> all()
}
```

MetadataRequest must also be modified. Topic name will be left in to allow requests to be made either by topic name or topic ID. Requests should only use one or the other.

ID will be checked first, but if the value is the default zero UUID, topic name will be used instead. If an ID is specified and the ID does not exist, the request will fail regardless of allow_auto_topic_creation.
If the topic ID is not found, the request will return an `UNKNOWN_TOPIC_ID` error for the topic indicating the topic ID did not exist. The check for the topic ID will occur before checking authorization on the topic. Thus, topic IDs are not considered sensitive information.

MetadataRequest v10

```
Metadata Request (Version: 10) => [topics] allow_auto_topic_creation include_cluster_authorized_operations
include_topic_authorized_operations
    topics => name topic_id*
      name => STRING (nullable)*
      topic_id* => UUID
    allow_auto_topic_creation => BOOL
    include_cluster_authorized_operations => BOOL
    include_topic_authorized_operations => BOOL
```

DeleteTopics

It will be useful for the AdminClient to be able to specify a list of topic Ids to delete to ensure the correct topics are being deleted. New methods will need to be added to the Admin interface and KafkaAdminClient

Admin and KafkaAdminClient

```
default DeleteTopicsResult deleteTopics(TopicCollection topics);

DeleteTopicsResult deleteTopics(TopicCollection topics, DeleteTopicsOptions options);
```

We also plan to deprecate the old methods in a future release. There are changes to DeleteTopicResult including deprecation of some of its old methods.

```
public class DeleteTopicsResult {

    protected DeleteTopicsResult(Map<Uuid, KafkaFuture<Void>> topicIdFutures, Map<String, KafkaFuture<Void>> 
nameFutures);

    protected static DeleteTopicsResult ofTopicIds(Map<Uuid, KafkaFuture<Void>> topicIdFutures);

    protected static DeleteTopicsResult ofTopicNames(Map<String, KafkaFuture<Void>> nameFutures);

    /**
     * @return a map from topic IDs to futures which can be used to check the status of
     * individual deletions if the deleteTopics request used topic IDs. Otherwise return null.
     */
    public Map<Uuid, KafkaFuture<Void>> topicIdValues()

    /**
     * @return a map from topic names to futures which can be used to check the status of
     * individual deletions if the deleteTopics request used topic names. Otherwise return null.
     */
    public Map<String, KafkaFuture<Void>> topicNameValues()

    @Deprecated
    /**
     * @return a map from topic names to futures which can be used to check the status of
     * individual deletions if the deleteTopics request used topic names. Otherwise return null.
     */
    public Map<String, KafkaFuture<Void>> values()

    /**
     * @return a future which succeeds only if all the topic deletions succeed.
     */
    public KafkaFuture<Void> all()
}
```

DeleteTopics Request and Response should be modified.

DeleteTopicsRequest v6

```
DeleteTopics Request (Version: 6) => [topics] timeout_ms
    topics => name topic_id*
      name => STRING (nullable)*
      topic_id* => UUID
    timeout_ms => INT32
```

Like the MetadataRequst, ID will be checked first, but if the value is the default zero UUID, topic name will be used instead. If an ID is specified and the ID does not exist, the request will return `UNKNOWN_TOPIC_ID` error for the topic indicating the topic ID did not exist. The check for the topic ID will occur before checking authorization on the topic. Thus, topic IDs are not considered sensitive information.

DeleteTopicsResponse v6

```
DeleteTopics Response (Version: 6) => throttle_time_ms [responses]
    throttle_time_ms => INT32
    responses => name topic_id* error_code error_message
      name => STRING (nullable)*
      topic_id* => UUID
      error_code => INT16
      error_message => STRING
```

Although only topic ID or only topic name are included in the request, if topic Ids are supported, the response will contain both the name and the ID.

Compatibility with KIP-500

KIP-500 and KIP-595 utilize a special metadata topic to store information that ZooKeeper has stored in the past. This topic must exist before the controller election, but in KIP-516, topic IDs are assigned in the controller. Here is an outline of how we can handle this.

Problem: KIP-595 describes a Vote Request which is used to elect the controller. Currently KIP-595 contains the topic name as part of the protocol.

Solution: Change Vote to use topic ID field. Use a sentinel ID reserved only for this topic before its ID is known.

Switching over to topic IDs in this KIP will result in fewer changes later on.

Problem: Post Zookeeper, a Fetch request for the metadata topic will be used to obtain information that was once stored in Zookeeper. KIP-516 stores topic IDs in Zookeeper, and the controller pushes them to brokers using LeaderAndIsrRequests. This will change to pulling the topic IDs to the broker with a fetch of the metadata topic. KIP-516 is replacing the topic name field with a topic ID field. So how will the first Fetch request know the correct topic ID for the metadata topic?

Solution: Use the same sentinel ID reserved for the metadata topic before its ID is known. After controller election, upon receiving the result, assign the metadata topic its unique topic ID. The ID should be written to the metadata topic, as all IDs will now be written to this topic instead of ZooKeeper.

Using a topic ID will result in a slightly smaller fetch request and likely prevent further changes. Assigning a unique ID for the metadata topic leaves the possibility for the topic to be placed in tiered storage, or used in other scenarios where topics from multiple clusters may be in one place without appending the cluster ID.

Sentinel ID

The idea is that this will be a hard-coded UUID that no other topic can be assigned. Initially the all zero UUID was considered, but was ultimately rejected since this is used as a null ID in some places and it is better to keep these usages separate. An example of a hard-coded UUID is `00000000000000000000000000000001`

LeaderAndIsr, UpdateMetadata, and StopReplica

As mentioned in KIP-631, LeaderAndIsr, UpdateMetadata, and StopReplica requests will become obsolete. However, most of the functionality of these requests will be replaced by using the Fetch Request on the metadata topic (as described in KIP-595). Since the fetch request will be updated to support topic IDs in this KIP, we will be able to accomplish the same goals.

Vote

Vote will be changed to replace topic name with topic ID, and will use a sentinel topic ID if no topic ID has been assigned already. See above for more information on sentinel topic IDs.

VoteRequest v0

```
VoteRequest (Version 0) => cluster_id [topics]
  cluster_id => STRING
  topics => topic_id* [partitions]
    topic_id* => UUID
    partitions => partition_index candidate_epoch candidate_id last_offset_epoch last_offset
      partition_index => INT32
      candidate_epoch => INT32
      candidate_id => INT32
      last_offset_epoch => INT32
      last_offset => INT64
```

VoteResponse v0

```
VoteResponse (Version 0) => error_code [topics]
  cluster_id => INT16
  topics => topic_id* [partitions]
    topic_id* => UUID
    partitions => partition_index error_code leader_id leader_epoch vote_granted
      partition_index => INT32
      error_code => INT16
      leader_id => INT32
      leader_epoch => INT32
      voted_granted => BOOL
```

BeginQuorumEpoch

BeginQuorumEpoch will replace the topic name field with the topic id field

BeginQuorumEpochRequest v0

```
BeginQuorumEpochRequest (Version 0) => cluster_id [topics]
  cluster_id => STRING
  topics => topic_id* [partitions]
    topic_id* => UUID
    partitions => partition_index leader_id leader_epoch
      partition_index => INT32
      leader_id => INT32
      leader_epoch => INT32
```

BeginQuorumEpochResponse v0

```
BeginQuorumEpochResponse (Version 0) => error_code [topics]
  cluster_id => INT16
  topics => topic_id* [partitions]
    topic_id* => UUID
    partitions => partition_index error_code leader_id leader_epoch
      partition_index => INT32
      error_code => INT16
      leader_id => INT32
      leader_epoch => INT32
```

EndQuorumEpoch

EndQuorumEpoch will replace the topic name field with the topic id field

EndQuorumEpochRequest v0

```
EndQuorumEpochRequest (Version 0) => cluster_id [topics]
  cluster_id => STRING
  topics => topic_id* [partitions]
    topic_id* => UUID
    partitions => partition_index replica_id leader_id leader_epoch [preferred_successors]
      partition_index => INT32
      replica_id => INT32
      leader_id => INT32
      leader_epoch => INT32
      preferred_successors => INT32
```

EndQuorumEpochResponse v0

```
EndQuorumEpochResponse (Version 0) => error_code [topics]
  cluster_id => INT16
  topics => topic_id* [partitions]
    topic_id* => UUID
    partitions => partition_index error_code leader_id leader_epoch
      partition_index => INT32
      error_code => INT16
      leader_id => INT32
      leader_epoch => INT32
```

log.dir layout

It would be ideal if the log.dir layout could be restructured from *{topic}_{partition}* format to {{topicIdprefix}}/*{topicId}_{partition}*, e.g. "mytopic_1" "24/24cc4332-f7de-45a3-b24e-33d61aa0d16c_1". Note the hierarchical directory structure using the first two characters of the topic ID to avoid having too many directories at the top level of the logdir. The exact formatting of the directory is not set in stone, but the idea is to replace topic names with topic IDs in the log directory. This is a significant change and will only be added upon a major release, most likely alongside KIP-500 changes that will also prevent downgrades.

Migration

Topic IDs will only be available to brokers with IBP version 2.8 or greater.

Upon a controller becoming active, the list of current topics is loaded from */brokers/topics/[topic].* When a topic without a topic ID is found, a UUID will be randomly generated and assigned the topic information at */brokers/topics/[topic]* will be updated with the id filled and the schema version bumped to version 3.

LeaderAndIsrRequest(s) will only be sent by the controller once a topic ID has been successfully assigned to the topic. Since the LeaderAndIsrRequest version was bumped, the IBP must also be bumped for migration.

When a replica receives a LeaderAndIsrRequest containing a topic ID for an existing partition which does not have a topic ID associated, it will create a partition metadata file for the topic partition locally. At this point the local partition will have been migrated to support topic IDs.

Compatibility, Deprecation, and Migration Plan

We will need to support all API calls which refer to a partition by either (topicId, partition) or (topicName, partition) until clients are updated to interact with topics by ID. In the first stages, deprecations are not currently planned.

However, when the directory structure is changed, downgrades will be no longer possible and the old directory structure will be deprecated.

Rejected Alternatives

Sequence ID

As an alternative to a topic UUID, a sequence number (long) could be maintained that is global for the given cluster.

This sequence number could be stored at */topicid/seqid*.

Upon topic creation, this sequence number will incremented, and the ID assigned to the created topic. Sequential topic ID generation can use the same approach to broker id generation.

If global uniqueness across clusters is required for topic IDs the first N bits of the ID could consist of a cluster ID prefix, followed by the sequence number. However, to achieve global uniqueness, this would require a large number of bits for the cluster ID prefix.

Use of a UUID has the benefit of being globally unique across clusters without partitioning the ID space by clusterID, and is conceptually simpler.

Topic Deletion

We considered and rejected two other strategies for performing topic deletes.

Best Effort Strategy

Under this stategy, the controller will attempt to send a StopReplicaRequest to all replicas. The controller will give up after a certain number of retries and will complete the delete. Although this will not simplify the topic deletion code, it will prevent delete topic requests from being blocked if one of the replicas is down. This would now be relatively safe, as stale topics will be deleted when a broker receives an initial LeaderAndIsrRequest, however it could prevent space from being reclaimed from a broker that does not respond to a StopReplicaRequest(s) before it is timed out, but is otherwise alive.

Send StopReplicaRequest(s) to online brokers only

In this approach, the controller will send StopReplicaRequests to only the brokers that are online, and will wait for a response from these brokers before marking the delete as successful. This will allow a topic delete to take place while some replicas are offline. If any replicas return to being online, they will receive an initial LeaderAndIsrRequest that will allow them to clear up any stale state. This is similar to the "best effort strategy above".

org.apache.kafka.common.TopicPartition

Eventually the TopicPartition class should include the topic ID. This may be difficult to enact until all APIs support topic IDs, and could come with a performance impact if implemented prior to this, as TopicPartitions are used for hashmap lookups throughout the broker.

Persisting Topic IDs

A few other alternatives to the partition metadata file were considered. One topic of discussion was whether it was necessary to include at all. While the the topic name is used in the directory structure, the only way to persist the topic ID to disk is through a file. As discussed above, the directory changes will not be added until a major release.

The file can also be used for tooling purposes, and may contain mappings that will be useful in the future.

Another alternative is to have a single file mapping all topic names to ids. Although this could be useful for tooling, it would be harder to maintain this file and update on each new topic added.

Future Work

Requests

The following requests could be improved by presence of topic IDs, but are out of scope for this KIP.

- CreatePartitionsRequest
- ElectPreferredLeadersRequest
- AlterReplicaLogDirsRequest
- AlterConfigsRequest
- DescribeConfigsRequest
- DescribeLogDirsRequest
- DeleteRecordsRequest
- AddPartitionsToTxnRequest
- TxnOffsetCommitRequest
- WriteTxnMarkerRequest

AdminClient

There are further changes to AdminClient made possible by adding topic ids. By adding topic ids to various request types (like those listed above) AdminClient can support identifying topics by ID. Some examples include but are not limited to:

- Using topic ids to specify what topics should receive new partitions in createPartitions
- Return ids in ListTopicsResult or TopicListing for listTopics
- Adding id to a type like TopicPartition or TopicPartitionReplica (see TopicIdPartition below)
    - Using topic ids (currently TopicPartition) to specify topic of the partitions for deleteRecords
    - Using topic ids (currently TopicPartitionReplica) to specify topic for alterReplicaLogDirs
    - Using topic ids (currently TopicPartition) to specify topic of the partitions for electLeaders

Clients

Some of the implemented request types are also relevant to clients. Adding full support for topic IDs in the clients would add an additional measure of safety when producing and consuming data. Fully supporting Topic IDs in clients is out of scope for this KIP due to the numerous public APIs that will need adjustments.

__consumer_offsets topic

Ideally, consumer offsets stored in the __consumer_offsets topic would be associated with the topic ID for which they were read. However, given the way the __consumer_offsets is compacted, this may be difficult to achieve in a forwards compatible way. This change will be left until topic IDs are implemented in the clients. Another future improvement opportunity is to use topicId in GroupMetadataManager.offsetCommitKey in the offset_commit topic. This may save some space.

Security/Authorization

One idea was to support authorizing a principal for a topic ID rather than a topic name. For now, this would be a breaking change, and it would be hard to support prefixed ACLs with topic IDs.

TopicIdPartition

Replacing a topic name with a topic ID introduced an issue with how the servers and clients should handle creating TopicPartition objects used when handling the requests. One id was to create a public TopicIdPartition object that could contain either a topic ID or a topic name along with the partition depending on the request version. However, TopicPartitions are used in numerous public APIs and all of those would have to be adapted to handle this new object. Due to the scope of these changes, the TopicIdPartition is out of scope and placed in future work.

Another alternative is adding an id field to the already existing TopicPartition object. However, even in this case, it is not yet clear all the implications, as some uses of TopicPartition require the topic name to be present.

Since this question requires a bit more thought and result in widespread changes, it is suggested that this change should be future work, perhaps its own KIP.